CSE 341
Lecture 29 a

JavaScript, the bad parts

slides created by Marty Stepp
http://www.cs.washington.edu/341/

see also:
JavaScript: The Good Parts, by Douglas Crockford
Bad parts related to variables

- **global variables and implied globals**
  
  ```javascript
  x = 7;  // oops, now i have a global x
  ```

- **lack of block scope**
  
  ```javascript
  if (x < 10) {
    var y = x + 3;
    print(y);
  }
  // oops, y still exists here
  ```

- **the global object and confusing uses of this**
  
  ```javascript
  this.x++;  // now it's 8. wait, what?
  ```
Bad parts: numbers

- `parseInt` is broken for some numbers:
  - `parseInt("032")` assumes it's octal, returns 26 (3*8+2)
  - `parseInt("08")` returns 0; 8 isn't a valid octal digit
  - solution: can pass a base
    - `parseInt("032", 10)` returns 32

- real number round-off
  - `0.1 + 0.2 // 0.30000000000000004`
  - many languages have this issue, but:
    - many novice programmers use JS, and this confuses them
    - for such a high-level lang., it is surprising to be stuck with it
Bad part: NaN

• NaN is a common numeric result with odd properties:
  
  3 * "x", 1 + null, undefined - undefined, ...

  ▪ hard to test for NaN:
    
    3 * "x" === NaN is false (nothing is equal to NaN)
    NaN === NaN is false!

  ▪ must use isNaN or isFinite function instead:
    
    isNaN(3 * "x") === true

• NaN and undefined are mutable; can be changed!

  undefined = 42;       // uh oh
  NaN = 1.0;           // Lulz
Bad parts: falsy values

• testing for the wrong falsy value can have bad results:

```javascript
function transferMoney(account) {
    // passes with 0, "", undefined, false, ...
    if (account.name == null) { ... }
}
```

• == is strange and produces odd results for falsy values:

```javascript
"" == false // true
0 == false // true
"0" == false // true
"\t\n" == 0 // false
null == undefined // true
" \t \n " == 0 // true
```
Bad part: semicolon insertion

- JS has a complex algorithm that allows you to omit semicolons and it will automatically insert them
  - nice for bad programmers who forget to use them
  - but often has weird and confusing results:

```javascript
// return an object
return
{
  name: "Joe",
  age: 15
}

// the code turns into...
return;
{
  name: "Joe",
  age: 15
}
```
Bad part: with

- the `with` statement runs code in context of an object:
  ```javascript
  var o = {name: "Bob", money: 2.50};
  with (o) {
      // now I don't have to say o.name
      if (name.length > 2) {
          money++; }
  }
  ```

- confusing when there's also a var named `name` or `money`
Bad part: eval

• the eval function compiles/executes a string as code:

```javascript
var s = "1 + 2 * 3";
var f = "function(s) { " +
           "print(s.toUppercase()); }";
var f = eval("f('hi');");
```

- seems nice, but it's slow, buggy, and bad for security
  – why is Scheme's eval better than this one?
Bad part: typeof

• typeof operator is broken for several types:
  ▪ for undefined: returns "undefined" (this is fine)
  ▪ for null: returns "object", not "null"
  ▪ for arrays: returns "object", not "array"
  ▪ for RegExps: returns "object" or "function"

• void is a JS operator that turns anything to undefined
  ▪ void("hello") returns "undefined"
    – useless, confusing to Java programmers
Bad part: Primitive wrappers

- numbers, booleans, strings are actually *primitives* in JS
  - but if they are used in an object-like way, they are silently temporarily converted into *wrapper* objects (~ like Java)
    - `(3).toString()` ← *creates temp object*

- you can explicitly construct wrappers, but don't ever do it:
  ```javascript
  var b = new Boolean(false);
  var n = new Number(42);
  var s = new String("hello");
  typeof(b) // "object"
  if (b) { print("hi"); } // does print!
  n === 42 // false
  ```
for (name in object) { statements; }

• "for-each" loops over each property's name in the object
  ▪ it also loops over the object's methods!

> for (prop in teacher) {
    print(prop + "=" + teacher[prop]); }

fullName=Marty Stepp
age=31
height=6.1
class=CSE 341
greet=function greet(you) {
    print("Hello " + you + ", I'm " + this.fullName);
}
Bad part: Never-empty objects

```javascript
var wordCount(text) {
    var counts = {}; // object 'map' of counters
    var words = text.split(/\s+/);
    for (var i = 0; i < words.length; i++) {
        if (counts[words[i]]) {
            counts[words[i]]++;
        } else {
            counts[words[i]] = 1;
        }
    }
    return counts;
}
```

- What if the text contains `this`, or `constructor`, or ...?
• Language design is hard and not to be taken lightly!
  ▪ every language has a few misguided or abusable features
  ▪ it's hard to change a language once it has been released
  ▪ sometimes adding features over time bloats a language
  ▪ add things coders need; don't add things coders don't need

  ▪ having more than 10 days to design a language is good
  ▪ having more than one person design a language is good
  ▪ mostly-copying another language can be very confusing